②

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS
BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
DTIC FILE COPY		

4. TITLE (and Subtitle)

Ada Compiler Validation Summary Report: R.R. Software, Inc., IntegrAda, 2.0.1, Zenith Z-184, (Host) and (Target), 880624W1.09123

5. TYPE OF REPORT & PERIOD COVERED

1 July 1988 to 1 July 1988

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)

Wright-Patterson AFB
Dayton, OH

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION AND

Wright-Patterson AFB
Dayton, OH

AD-A207 031

10. PROGRAM ELEMENT. PROJECT. TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Ada Joint Program Office
United States Department of Defense
Washington, DC 20301-3081

12. REPORT DATE

13. NUMBER OF PAGES

14. MONITORING AGENCY NAME & ADDRESS(If different from Controlling Office)

Wright-Patterson AFB
Dayton, OH

15. SECURITY CLASS (of this report)
UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20 If different from Report)

UNCLASSIFIED

DTIC
ELECTED
14 APR 1989
E

18. SUPPLEMENTARY NOTES

19. KEYWORDS (Continue on reverse side if necessary and identify by block number)

Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
R.R. Software, Inc., IntegrAda, 2.0.1, Wright-Patterson AFB, Zenith Z-184 under Zenith MS-DOS, 3.21 (Host) to Zenith Z-184 under Zenith MS-DOS, 3.21 (Target), ACVC 1.9.

89 4 11 072

‘

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 880624W1.09123
R.R. Software, Inc.
IntegrAda, 2.0.1
Zenith Z-184

DTIC
COPY
INSPECTED
4

Completion of On-Site Testing:
1 July 1988

Prepared By:
Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Accession For

NTIS GRA&I	X
DTIC TAB	☐
Unannounced	☐
Justification	

By
Distribution/

Availability Codes

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington DC 20301-3081

| Dist | Avail and/or Special |
| A-1 | |

Ada Compiler Validation Summary Report:

Compiler Name: IntegrAda, 2.0.1

Certificate Number: 880624W1.09123

Host: Target:
    Zenith Z-184 under   Zenith Z-184 under
    Zenith MS-DOS, 3.21   Zenith MS-DOS, 3.21

Testing Completed 1 July 1988 Using ACVC 1.9

This report has been reviewed and is approved.

Steven C. Wilson
Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH   45433-6503
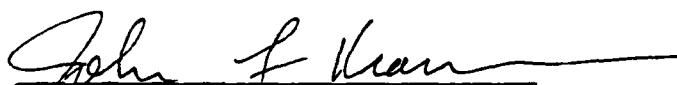
John F. Kramer
Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria VA   22311
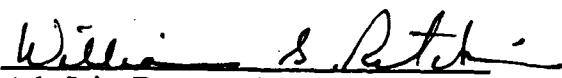
William S. Ritchie
Ada Joint Program Office
William S. Ritchie
Acting Director
Department of Defense
Washington, DC 20301

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent but permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

1-1

INTRODUCTION

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard

- To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard

- To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by SofTech, Inc. under the direction of the AVF according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO). On-site testing was completed 1 July 1988 at Madison, WI.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

> Ada Information Clearinghouse
> Ada Joint Program Office
> OUSDRE
> The Pentagon, Rm 3D-139 (Fern Street)
> Washington DC 20301-3081

or from:

> Ada Validation Facility
> ASD/SCEL
> Wright-Patterson AFB OH 45433-6503

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

> Ada Validation Organization
> Institute for Defense Analyses
> 1801 North Beauregard Street
> Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language,
   ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

2. Ada Compiler Validation Procedures and Guidelines, Ada Joint
   Program Office, 1 January 1987.

3. Ada Compiler Validation Capability Implementers' Guide, SofTech,
   Inc., December 1986.

4. Ada Compiler Validation Capability User's Guide, December 1986.

1.4 DEFINITION OF TERMS

ACVC   The Ada Compiler Validation Capability. The set of Ada
                programs that tests the conformity of an Ada compiler to the
                Ada programming language.

Ada   An Ada Commentary contains all information relevant to the
Commentary   point addressed by a comment on the Ada Standard. These
                comments are given a unique identification number having the
                form AI-ddddd.

Ada Standard   ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

Applicant   The agency requesting validation.

AVF   The Ada Validation Facility. The AVF is responsible for
                conducting compiler validations according to procedures
                contained in the Ada Compiler Validation Procedures and
                Guidelines.

AVO   The Ada Validation Organization. The AVO has oversight
                authority over all AVF practices for the purpose of
                maintaining a uniform process for validation of Ada
                compilers. The AVO provides administrative and technical

support for Ada validations to ensure consistent practices.

Compiler
A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.

Failed test
An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.

Host
The computer on which the compiler resides.

Inapplicable test
An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.

Passed test
An ACVC test for which a compiler generates the expected result.

Target
The computer for which a compiler generates code.

Test
A program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.

Withdrawn test
An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce compilation or link errors.

Class A tests check that legal Ada programs can be successfully compiled and executed. There are no explicit program components in a Class A test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of the tests in the ACVC follow conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and

place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the
following configuration:

Compiler: IntegrAda, 2.0.1

ACVC Version: 1.9

Certificate Number: 880624W1.09123

Host Computer:

Machine:	Zenith Z-184
Operating System:	Zenith MS-DOS, 3.21
Memory Size:	640K

Target Computer:

Machine:	Zenith Z-184
Operating System:	Zenith MS-DOS, 3.21
Memory Size:	640K

2.2   IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ.   Class D and E tests specifically check for such implementation differences.   However, tests in other classes also characterize an implementation.   The tests demonstrate the following characteristics:

- Capacities.

  The compiler correctly processes tests containing loop statements nested to 17 levels, and recursive procedures separately compiled as subunits nested to 6 levels. The use of 65 levels of block nesting exceeds the capacity of the compiler.   It correctly processes a compilation containing 723 variables in the same declarative part.   (See tests D55A03A..H (8 tests), D56001B, D64005E..G (3 tests), and D29002K.)

- Universal integer calculations.

  An implementation is allowed to reject universal integer calculations having values that exceed SYSTEM.MAX_INT.   This implementation processes 64 bit integer calculations.   (See tests D4A002A, D4A002B, D4A004A, and D4A004B.)

- Predefined types.

  This implementation supports the additional predefined type LONG_FLOAT in the package STANDARD.   (See tests B86001C and B86001D.)

- Based literals.

  An implementation is allowed to reject a based literal with a value exceeding SYSTEM.MAX_INT during compilation, or it may raise NUMERIC_ERROR or CONSTRAINT_ERROR during execution.   This implementation raises NUMERIC_ERROR during execution.   (See test E24101A.)

- Expression evaluation.

  Apparently some default initialization expressions for record components are evaluated before any value is checked to belong to a component's subtype.   (See test C32117A.)

  Assignments for subtypes are performed with the same precision as the base type.   (See test C35712B.)

This implementation uses no extra bits for extra precision. This implementation uses all extra bits for extra range. (See test C35903A.)

Sometimes NUMERIC_ERROR is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

No exception is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type.(See test C45252A.)

Apparently underflow is gradual. (See tests C45524A..Z.)

. Rounding.

The method used for rounding to integer is apparently round away from zero. (See tests C46012A..Z.)

The method used for rounding to longest integer is apparently round away from zero. (See tests C46012A..Z.)

The method used for rounding to integer in static universal real expressions is apparently round away from zero. (See test C4A014A.)

. Array types.

An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT. For this implementation:

Declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises no exception. (See test C36003A.)

NUMERIC_ERROR is raised when 'LENGTH is applied to an array type with INTEGER'LAST + 2 components. (See test C36202A.)

NUMERIC_ERROR is raised when 'LENGTH is applied to an array type with SYSTEM.MAX_INT + 2 components. (See test C36202B.)

A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises STORAGE_ERROR when the array objects are declared. (See test C52103X.)

A packed two-dimensiona BOOLEAN array with more than INTEGER'LAST components raises CONSTRAINT_ERROR when the length of a dimension is calculated and exceeds INTEGER'LAST. (See test C52104Y.)

A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises no exception. (See test E52103Y.)

In assigning one-dimensional array types, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. In assigning two-dimensional array types, the expression does not appear to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

. Discriminated types.

During compilation, an implementation is allowed to either accept or reject an incomplete type with discriminants that is used in an access type definition with a compatible discriminant constraint. This implementation accepts such subtype indications. (See test E38104A.)

In assigning record types with discriminants, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

. Aggregates.

In the evaluation of a multi-dimensional aggregate, index subtype checks appear to be made as choices are evaluated. (See tests C43207A and C43207B.)

In the evaluation of an aggregate containing subaggregates, not all choices are evaluated before being checked for identical bounds. (See test E43212B.)

Not all choices are evaluated before CONSTRAINT_ERROR is raised if a bound in a nonnull range of a nonnull aggregate does not belong to an index subtype. (See test E43211B.)

. Representation clauses.

An implementation might legitimately place restrictions on representation clauses used by some of the tests. If a representation clause is used by a test in a way that violates a restriction, then the implementation must reject it.

Enumeration representation clauses containing noncontiguous values for enumeration types other than character and boolean types are not supported. (See tests C35502I..J, C35502M..N, and A39005F.)

Enumeration representation clauses containing noncontiguous values for character types are not supported. (See tests C35507I..J, C35507M..N, and C55B16A.)

Enumeration representation clauses for boolean types containing representational values other than (FALSE => 0, TRUE => 1) are not supported. (See tests C35508I..J and C35508M..N.)

Length clauses with SIZE specifications for enumeration types are supported. (See test A39005B.)

Length clauses with STORAGE_SIZE specifications for access types are not supported. (See tests A39005C and C87B62B.)

Length clauses with STORAGE_SIZE specifications for task types are not supported. (See tests A39005D and C87B62D.)

Length clauses with SMALL specifications are not supported. (See tests A39005E and C87B62C.)

Record representation clauses are not supported. (See test A39005G.)

Length clauses with SIZE specifications for derived integer types are supported. (See test C87B62A.)

. Pragmas.

The pragma INLINE is not supported for procedures. The pragma INLINE is not supported for functions. (See tests LA3004A, LA3004B, EA3004C, EA3004D, CA3004E, and CA3004F.)

. Input/output.

The package SEQUENTIAL_IO and DIRECT_IO cannot be instantiated with unconstrained array types. (See tests EE2201D and EE2401D.)

Modes IN_FILE and OUT_FILE are supported for SEQUENTIAL_IO and DIRECT_IO. (See tests CE2102D and CE2102E)

Modes IN_FILE, OUT_FILE, and INOUT_FILE are supported for DIRECT_IO. (See tests CE2102F, CE2102I, and CE2102J.)

RESET and DELETE are supported for SEQUENTIAL_IO and DIRECT_IO. (See tests CE2102G and CE2102K.)

Dynamic creation and deletion of files are supported for SEQUENTIAL_IO and DIRECT_IO. (See tests CE2106A and CE2106B.)

Overwriting to a sequential file does not truncate the file. (See test CE2208B.)

An existing text file can be opened in OUT_FILE mode, can be created in OUT_FILE mode, and can be created in IN_FILE mode. (See test EE3102C.)

More than one internal file can be associated with each external file for text I/O for reading only. (See tests CE3111A..E (5 tests), CE3114B, and CE3115A.)

More than one internal file can be associated with each external file for sequential I/O for reading only. (See tests CE2107A..D (4 tests), CE2110B, and CE2111D.)

More than one internal file can be associated with each external file for direct I/O for reading only. (See tests CE2107F..I (5 tests), CE2110B, and CE2111H.)

An internal sequential access file and an internal direct access file cannot be associated with a single external file for writing. (See test CE2107E.)

An external file associated with more than one internal file cannot be deleted for SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO. (See test CE2110B.)

Temporary sequential files are given names. Temporary direct files are given names. Temporary files given names are not deleted when they are closed. (See tests CE2108A and CE2108C.)

• Generics.

Generic subprogram declarations and bodies can be compiled in separate compilations. (See tests CA1012A and CA2009F.)

Generic package declarations and bodies can be compiled in separate compilations. (See tests CA2009C, BC3204C, and BC3205D.)

Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.9 of the ACVC comprises 3122 tests. When this compiler was tested, 27 tests had been withdrawn because of test errors. The AVF determined that 290 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation. Modifications to the code, processing, or grading for 30 tests were required to successfully demonstrate the test objective. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	105	1046	1586	10	14	44	2805
Inapplicable	5	5	267	7	4	2	290
Withdrawn	3	2	21	0	1	0	27
TOTAL	113	1053	1874	17	19	46	3122

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	2	3	4	5	6	7	8	9	10	11	12	13	14	TOTAL
Passed	190	483	528	236	164	98	140	326	131	36	234	3	236	2805
Inapplicable	14	89	146	12	2	0	3	1	6	0	0	0	17	290
Withdrawn	2	14	3	0	0	1	2	0	0	0	2	1	2	27
TOTAL	206	586	677	248	166	99	145	327	137	36	236	4	255	3122

3.4 WITHDRAWN TESTS

The following 27 tests were withdrawn from ACVC Version 1.9 at the time of this validation:

B28003A	E28005C	C34004A	C35502P	A35902C
C35904A	C35904B	C35A03E	C35A03R	C37213H
C37213J	C37215C	C37215E	C37215G	C37215H
C38102C	C41402A	C45332A	C45614C	A74106C
C85018B	C87B04B	CC1311B	BC3105A	AD1A01A
CE2401H	CE3208A			

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 290 tests were inapplicable for the reasons indicated:

C35502I..J (2 tests), C35502M..N (2 tests), C35507I..J (2 tests), C35507M..N (2 tests), C35508I..J (2 tests), C35508M..N (2 tests), A39005F, and C55B16A use enumeration representation clauses which are not supported by this compiler.

- C35702A uses SHORT_FLOAT which is not supported by this implementation.

- A39005C and C87B62B use length clauses with STORAGE_SIZE specifications for access types which are not supported by this implementation.

- A39005D and C87B62D use STORAGE_SIZE specifications on task types which are not supported by this implementation.

- A39005E uses length clauses with SMALL specifications which are not supported by this implementation.

- A39005G uses a record representation clause which is not supported by this compiler.

- The following tests use SHORT_INTEGER, which is not supported by this compiler:

C45231B	C45304B	C45502B	C45503B	C45504B
C45504E	C45611B	C45613B	C45614B	C45631B
C45632B	B52004E	C55B07B	B55B09D	

- The following tests use LONG_INTEGER, which is not supported by this compiler:

C45231C	C45304C	C45502C	C45503C	C45504C
C45504F	C45611C	C45613C	C45631C	C45632C
B52004D	C55B07A	B55B09C		

- C45231D requires a macro substitution for any predefined numeric types other than INTEGER, SHORT_INTEGER, LONG_INTEGER, FLOAT, SHORT_FLOAT, and LONG_FLOAT. This compiler does not support any such types.

- C45531M, C45531N, C45532M, and C45532N use fine 48-bit fixed-point base types which are not supported by this compiler.

- C45531O, C45531P, C45532O, and C45532P use coarse 48-bit fixed-point base types which are not supported by this compiler.

- D55A03E..H (4 tests) use more than 17 levels of loop nesting which exceeds the capacity of the compiler.

- D56001B uses 65 levels of block nesting which exceeds the capacity of the compiler.

- D64005F and D64005G use nested procedures as subunits to a level of 10 which exceeds the capacity of the compiler.

- B86001D requires a predefined numeric type other than those defined by the Ada language in package STANDARD. There is no such type for this implementation.

- C96005B requires the range of type DURATION to be different from those of its base type; in this implementation they are the same.

- CA3004E, EA3004C, and LA3004A use the INLINE pragma for procedures, which is not supported by this compiler.

- CA3004F, EA3004D, and LA3004B use the INLINE pragma for functions, which is not supported by this compiler.

- EE2201D and EE2401D use instantiations of package SEQUENTIAL_IO and DIRECT_IO with unconstrained array types. These instantiations are rejected by this compiler.

- CE2107B..E (4 tests), CE2107G..I (3 tests), CE2110B, CE2111D, CE2111H, CE3111B..E (4 tests), and CE3114B are inapplicable because multiple internal files cannot be associated with the same external file if any file is open for writing. The proper exception is raised when multiple access is attempted.

- The following 201 tests require a floating-point accuracy that exceeds the maximum of 15 digits supported by this implementation:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into subtests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that wasn't anticipated by the test (such as raising one exception instead of another).

Modifications were required for 30 Class B tests.

The following Class B tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B22003A	B24007A	B24009A	B25002A	B26005A
B27005A	B29001A	B37106A	B37201A	B44001A

TEST INFORMATION

B49003A	B49005A	B51001A	B55A01A	B63001A
B63001B	B64001A	B91001H	B95001A	BA1101A
BA1101C	BA1101E	BA3006A	BA3006B	BA3007B
BA3008A	BA3008B	BA3013A	BC2001D	BC2001E

C45651A requires that the result of the expression in line 227 be in the range given in line 228; however, this range excludes some acceptable results. This implementation passes all other checks of this test, and the AVO ruled that the test is passed.

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.9 produced by the IntegrAda, 2.0.1 compiler was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the IntegrAda, 2.0.1 compiler using ACVC Version 1.9 was conducted on-site by a validation team from the AVF. The configuration consisted of a Zenith Z-184 operating under Zenith MS-DOS, 3.21.

A magnetic tape containing all tests except for withdrawn tests and tests requiring unsupported floating-point precisions was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the magnetic tape. Tests requiring modifications during the prevalidation testing were included in their modified form on the magnetic tape.

The contents of the magnetic tape were loaded directly onto the host computer. After the test files were loaded to disk, the full set of tests was compiled and linked on the Zenith Z-184, and all executable tests were run. Object files were linked and executed on the target. Results were printed from the target computer.

The compiler was tested using command scripts provided by R.R. Software, Inc. and reviewed by the validation team. The compiler was tested using all default option settings except for the following:

Option	Effect
/W	Suppress warning messages.
/Q	Suppress prompts at error messages.
/T	Generate trimmable code.
/D	Debugging code off.
/E	Generate an EXE file.
/O1	Use memory model 1.

Tests were compiled, linked, and executed (as appropriate) using a single host and target computer. Test output, compilation listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

3.7.3 Test Site

Testing was conducted at Madison, WI and was completed on 1 July 1988.

APPENDIX A

DECLARATION OF CONFORMANCE

R.R. Software, Inc. has submitted the following
Declaration of Conformance concerning the IntegrAda,
2.0.1.

DECLARATION OF CONFORMANCE

Compiler Implementor: R.R. Software, Inc.
Ada Validation Facility: ASD/SCEL, Wright-Patterson AFB OH 45433-6503
Ada Compiler Validation Capability (ACVC) Version: 1.9

Base Configuration

Base Compiler Name: IntegrAda   Version: 2.0.1
Host Architecture ISA: Zenith Z-184   OS&VER #: Zenith MS-DOS, 3.21
Target Architecture ISA: Zenith Z-184 OS&VER #: Zenith MS-DOS, 3.21

Implementor's Declaration

I, the undersigned, representing R.R. Software, Inc., have implemented no
deliberate extensions to the Ada Language Standard ANSI/MIL-STD-1815A in
the compiler(s) listed in this declaration. I declare that R.R. Software,
Inc. is the owner of record of the Ada language compiler(s) listed above
and, as such, is responsible for maintaining said compiler(s) in
conformance to ANSI/MIL-STD-1815A. All certificates and registrations for
Ada language compiler(s) listed in this declaration shall be made only in
the owner's corporate name.
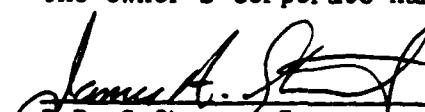
R.R. Software, Inc.   Date: July 1, 1988
James A. Stewart, General Manager
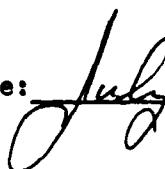
Owner's Declaration

I, the undersigned, representing   AETECH,   take   full
responsibility for implementation and maintenance of the Ada compiler(s)
listed above, and agree to the public disclosure of the final Validation
Summary Report. I further agree to continue to comply with the Ada
trademark policy, as defined by the Ada Joint Program Office. I declare
that all of the Ada language compilers listed, and their host/target
performance, are in compliance with the Ada Language Standard
ANSI/MIL-STD-1815A.

James Thomes, President   Date:
AETECH

A-2
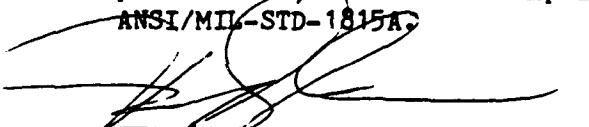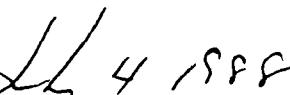
APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the IntegrAda, 2.0.1 compiler, are described in the following sections, which discuss topics in Appendix F of the Ada Standard. Implementation-specific portions of the package STANDARD are also included in this appendix.

```
package STANDARD is

    ...

    type INTEGER is range -32768 .. 32767;

    type FLOAT is digits 6 range -((2.0 ** 128) - (2.0 ** 104)) ..
                                  ((2.0 ** 128) - (2.0 ** 104));

    type LONG_FLOAT is digits 15 range -((2.0 ** 1024) - (2.0 ** 971)) ..
                                        ((2.0 ** 1024) - (2.0 ** 971));

    type DURATION is delta 1.0/4096.0 range -((2.0 ** 31) - 1)/4096.0 ..
                                              ((2.0 ** 31) - 1)/4096.0;

    ...

end STANDARD;
```

F   Implementation Dependencies

This appendix specifies certain system-dependent characteristics of IntegrAda.

F.1   Implementation Dependent Pragmas

In addition to the required Ada pragmas, IntegrAda also provides several others. Some of these pragmas have a *textual range*. Such pragmas set some value of importance to the compiler, usually a flag that may be On or Off. The value to be used by the compiler at a given point in a program depends on the parameter to the most recent relevant pragma in the text of the program. For flags, if the parameter is the identifier On, then the flag is on; if the parameter is the identifier Off, then the flag is off; if no such pragma has occurred, then a default value is used.

The range of a pragma – even a pragma that usually has a textual range – may vary if the pragma is not inside a compilation unit. This matters only if you put multiple compilation units in a file. The following rules apply:
1)   If a pragma is inside a compilation unit, it affects only that unit.
2)   If a pragma is outside a compilation unit, it affects all following compilation units in the compilation.

Certain required Ada pragmas, such as INLINE, would follow different rules; however, as it turns out, IntegrAda ignores all pragmas that would.

The following system-dependent pragmas are defined by IntegrAda. Unless otherwise stated, they may occur anywhere that a pragma may occur.

ALL_CHECKS Takes one of two identifiers On or Off as its argument, and has a textual range. If the argument is Off, then this pragma causes suppression of arithmetic checking (like pragma ARITHCHECK – see below), range checking (like pragma RANGECHECK – see below), storage error checking, and elaboration checking. If the argument is On, then these checks are all performed as usual. Note that pragma ALL_CHECKS does not affect the status of the DEBUG pragma; for the fastest run time code (and the worst run time checking), both ALL_CHECKS and DEBUG should be turned Off and the pragma OPTIMIZE (Time) should be used. Note also that ALL_CHECKS does not affect the status of the ENUMTAB pragma. Combining check suppression using the pragma ALL_CHECKS and using the pragma SUPPRESS may cause unexpected results; it should not be done. However, ALL_CHECKS may be combined with the IntegrAda pragmas ARITHCHECK and RANGECHECK; whichever relevant pragma has

occurred most recently will determine whether a given check is performed. ALL_CHECKS is on by default. Turning any checks off may cause unpredictable results if execution would have caused the corresponding assumption to be violated. Checks should be off only in fully debugged and tested programs. After checks are turned off, full testing should again be done, since any program that handles an exception may expect results that will not occur if no checking is done.

ARITHCHECK Takes one of the two identifiers On or Off as its argument, and has a textual range. Where ARITHCHECK is on, the compiler is permitted to (and generally does) not generate checks for situations where it is permitted to raise NUMERIC_ERROR; these checks include overflow checking and checking for division by zero. Combining check suppression using the pragma ARITHCHECK and using the pragma SUPPRESS may cause unexpected results; it should not be done. However, ARITHCHECK may be combined with the IntegrAda pragma ALL_CHECKS; whichever pragma has occurred most recently will be effective. ARITHCHECK is on by default. Turning any checks off may cause unpredictable results if execution would have caused the corresponding assumption to be violated. Checks should be off only in fully debugged and tested programs. After checks are turned off, full testing should again be done, since any program that handles an exception may expect results that will not occur if no checking is done.

CLEANUP Takes an integer literal in the range 0..3 as its argument, and has a textual range. Using this pragma allows the IntegrAda run-time system to be less than meticulous about recovering temporary memory space it uses. This pragma can allow for smaller and faster code, but can be dangerous: certain constructs can cause memory to be used up very quickly. The smaller the parameter, the more danger is permitted. A value of three – the default value – causes the run-time system to be its usual immaculate self. A value of zero causes no reclamation of temporary space. Values of one and two allow compromising between cleanliness and speed. Using values other than 3 adds some risk of your program running out of memory, especially in loops which contain certain constructs.

DEBUG Takes one of the two identifiers On or Off as its argument, and has a textual range. This pragma controls the generation of line number code and procedure name code. When DEBUG is on, such code is generated. When DEBUG is off, no line number code or procedure names are generated. This information is used by the walkback which

is generated after a run-time error (e.g., an unhandled exception). The walkback is still generated when DEBUG is off, but the line numbers will be incorrect, and no subprogram names will be printed. DEBUG's initial state can be set by the command line; if no explicit option is given, then DEBUG is initially on. Turning DEBUG off saves space, but causes the loss of much of IntegrAda's power in describing run time errors.

Notes:
DEBUG should only be turned off when the program has no errors. The information provided on an error when DEBUG is off is not very useful.

If DEBUG is on at the beginning of a subprogram or package specification, then it must be on at the end of the specification. Conversely, if DEBUG is off at the beginning of such a specification, it must be off at the end. If you want DEBUG to be off for an entire compilation, then you can either put a DEBUG pragma in the context clause of the compilation or you can use the appropriate compiler option.

ENUMTAB   Takes one of the two identifiers On or Off as its argument, and has a textual range. This pragma controls the generation of enumeration tables. Enumeration tables are used for the attributes IMAGE, VALUE, and WIDTH, and hence to input and output enumeration values. The tables are generated when ENUMTAB is on. The state of the ENUMTAB flag is significant only at enumeration type definitions. If this pragma is used to prevent generation of a type's enumeration tables, then using the three mentioned attributes causes an erroneous program, with unpredictable results; furthermore, the type should not be used as a generic actual discrete type, and in particular TEXT_IO.ENUMERATION_IO should not be instantiated for the type. If the enumeration type is not needed for any of these purposes, the tables, which use a lot of space, are unnecessary. ENUMTAB is on by default.

PAGE_LENGTH
            This pragma takes a single integer literal as its argument. It says that a page break should be added to the listing after the each occurrence of the given number of lines. The default page length is 32000, so that no page breaks are generated for most programs. Each page starts with a header that looks like the following:
                IntegrAda Version 2.0.0 compiling *file* on *date* at *time*

RANGECHECK Takes one of the two identifiers On or Off as its argument, and has a textual range. Where RANGECHECK is on, the compiler is permitted to (and generally does) not generate checks for situations where it is expected to raise CONSTRAINT_ERROR; these checks include null pointer checking, discriminant checking, index checking, array length checking, and range checking. Combining check suppression using the pragma RANGECHECK and using the pragma SUPPRESS may cause unexpected results; it should not be done. However, RANGECHECK may be combined with the IntegrAda pragma ALL_CHECKS; whichever pragma has occurred most recently will be effective. RANGECHECK is on by default. Turning any checks off may cause unpredictable results if execution would have caused the corresponding assumption to be violated. Checks should be off only in fully debugged and tested programs. After checks are turned off, full testing should again be done, since any program that handles an exception may expect results that will not occur if no checking is done.

SYSLIB This pragma tells the compiler that the current unit is one of the standard IntegrAda system libraries. It takes as a parameter an integer literal in the range 1 .. 15; only the values one through four are currently used. For example, system library number two provides floating point support. Do not use this pragma unless you are writing a package to replace one of the standard IntegrAda system libraries.

VERBOSE Takes On or Off as its argument, and has a textual range. VERBOSE controls the amount of output on an error. If VERBOSE is on, the 2 lines preceding the error are printed, with an arrow pointing at the error. If VERBOSE is off, only the line number is printed.

VERBOSE(Off):

```
Line 16 at Position 5
*ERROR* Identifier is not defined
```

VERBOSE(On):

```
15: if X = 10 then
16:     Z := 10;
----------
*ERROR* Identifier is not defined
```

The reason for this option is that an error message with VERBOSE on can take a long time to be generated, especially in a large program. VERBOSE's initial condition can be set by the compiler command line.

Several required Ada pragmas may have surprising effects in IntegrAda. The PRIORITY pragma may only take the value 0, since that is the only value in the range System.Priority. Specifying any OPTIMIZE pragma turns on optimization; otherwise, optimization is only done if specified on the compiler's command line. The SUPPRESS pragma is ignored unless it only has one parameter. Also, the following pragmas are always ignored: CONTROLLED, INLINE, INTERFACE, MEMORY_SIZE, PACK, SHARED, STORAGE_UNIT, and SYSTEM_NAME. Pragma CONTROLLED is always ignored because IntegrAda does no automatic garbage collection; thus, the effect of pragma CONTROLLED already applies to all access types. Pragma SHARED is similarly ignored: IntegrAda's non-preemptive task scheduling gives the appropriate effect to all variables. The pragmas INLINE, PACK, and SUPPRESS (with two parameters) all provide recommendations to the compiler; as Ada allows, the recommendations are ignored. No other languages are supported that use the INTERFACE pragma. The pragmas MEMORY_SIZE, STORAGE_UNIT, and SYSTEM_NAME all attempt to make changes to constants in the System package; in each case, IntegrAda allows only one value, so that the pragma is ignored.

F.2   Implementation Dependent Attributes

IntegrAda does not provide any attributes other than the required Ada attributes.

Some of the required Ada attributes provide system-dependent information; some of the interesting cases are listed below.

The Address attribute in IntegrAda returns a value of the type System.Address, which refers to data segment addresses. For subprograms, packages, task types, and labels, the conventional value 0 is returned (since these addresses are outside the data segment). If the value returned by the address attribute is less than zero, it refers to an address that is 65536 greater than the given value; that is, the address can be considered to be a whole number in the standard 8086 format.

The Size attribute gives the size of the non-dynamic part of an object, type, or subtype. For an array with non-static bounds, for example, the Size attribute returns the size of the array descriptor.

The attribute Storage_Size for an access type always returns the universal integer value 65536 (the size of the data segment). This occurs because, in theory, the values of an access type may take up all of the data segment. In practice, some of the data segment will be taken up by other data.

F.3   Specification of the Package SYSTEM

The package System for IntegrAda has the following definition.

```
package System is

    -- System package for IntegrAda

    type Address is new Integer;
    type Name is (MS_DOS2);

    System_Name : constant Name := MS_DOS2;

    Storage_Unit : constant := 8;
    Memory_Size : constant := 65536;
        -- Note: The actual memory size of a program is determined
        -- dynamically; this is the maximum number of bytes in the data
        -- segment.

    -- System Dependent Named Numbers:
    Min_Int : constant := -32768;
    Max_Int : constant := 32767;
    Max_Digits : constant := 15;
    Max_Mantissa : constant := 31;
    Fine_Delta : constant := 2#1.0#E-31;
        -- equivalently, 4.656612873077392578125E-10
    Tick : constant := 0.01; -- Some machines have less accuracy;
        -- for example, the IBM PC actually ticks about
        -- every 0.06 seconds.

    -- Other System Dependent Declarations
    subtype Priority is Integer range 0..0;

    type Byte is specially defined, see below;

end System;
```

The type Byte in the System package corresponds to the 8-bit machine byte.

The type System.Byte can be considered to be an enumeration type with no visible literals. The type is discrete, so that values of the type may be obtained using the Val attribute. The parameter to the Val attribute must have a value between 0 and 255; if it is not, the exception CONSTRAINT_ERROR will be raised.

Since Byte is a discrete type, it can be used as the type of an array index, a loop parameter, a case expression, and so on. It is not a numeric type, so the predefined numeric operators cannot be used on objects of the type.

F.4   Restrictions on Representation Clauses

IntegrAda representation clauses are currently rather unpretentious. Specifically, IntegrAda currently only allows certain representation clauses that simply echo what the compiler would have chosen anyway. This minimal implementation of representation clauses helps the IntegrAda compiler to be fast and to fit in the limited memory address space of various machines.

Specifically, there are the following restrictions:

A length clause that specifies T'SIZE for a type T must give the default size for T.

A length clause that specifies T'STORAGE_SIZE is not supported; IntegrAda uses a single large common heap.

A length clause that specifies T'STORAGE_SIZE for a task type T is currently not supported. This feature is scheduled for support in the next version of IntegrAda. The given value of T'STORAGE_SIZE specifies an amount of stack space to be used by tasks of the given type; any allocated objects are allocated in a different area of the data segment.

A length clause that specifies T'SMALL for a fixed point type must give the default value of T'SMALL, namely the greatest power of two less than or equal to the delta specified for the type. This value must be in the range
$$2.0 ** (-99) .. 2.0 ** 99,$$
inclusive.

An enumeration representation clause for a type T must map the values of the type T to consecutive integers starting with zero.

The expression in an alignment clause in a record representation clause must equal one.

A component clause must give a storage place that is equivalent to the default value of the POSITION attribute for such a component.

Appendix F: Implementation Dependencies

A component clause must give a range that starts at zero and extends to one less than the size of the component.

IntegrAda does not support any address clauses; hence, IntegrAda does not support any interrupt entries.

The rules for representation clauses, together with the fact that the pragma PACK is ignored in IntegrAda, imply that type conversions cannot cause a change of representation in IntegrAda.

F.5   Implementation Defined Names

IntegrAda uses no implementation generated names.

F.6   Address Clause Expressions

IntegrAda does not support any address clauses.

F.7   Unchecked_Conversion Restrictions

We first make the following definitions:
A type or subtype is said to be a *simple type* or a *simple subtype* (respectively) if it is a scalar (sub)type, an access (sub)type, a task (sub)type, or if it satisfies the following two conditions:
1) If it is an array type or subtype, then it is constrained and its index constraint is static; and
2) If it is a composite type or subtype, then all of its subcomponents have a simple subtype.

A (sub)type which does not meet these conditions is called *non-simple*. Discriminated records can be simple; variant records can be simple. However, constraints which depend on discriminants are non-simple (because they are non-static).

IntegrAda imposes the following restriction on instantiations of Unchecked_Conversion: for such an instantiation to be legal, both the source actual subtype and the target actual subtype must be simple subtypes, and they must have the same size.

F.8   Implementation Dependencies of I/O

The syntax of a external file name depends on the operating system being used. Some external files do not really specify disk files; these are called *devices*. Devices are specified by special file names, and are treated specially by some of the I/O routines.

The syntax of an MS-DOS 2.xx or 3.xx filename is:

    [d:][path]filename[.ext]

where "d:" is an optional disk name; "path" is an optional path consisting of directory names, each followed by a backslash; "filename" is the filename (maximum 8 characters); and ".ext" is the extension (or file type). See your MS-DOS manual for a complete description. In addition, the following special device names are recognized:

STI: MS-DOS standard input. The same as Standard_Input. Input is buffered by lines, and all MS-DOS line editing characters may be used. Can only be read.
STO: MS-DOS standard output. The same as Standard_Output. Can only be written.
ERR: MS-DOS standard error. The output to this device cannot be redirected. Can only be written.
CON: The console device. Single character input with echoing. Due to the design of MS-DOS, this device *can* be redirected. Can be read and written.
AUX: The auxiliary device. Can be read or written.
LST: The list (printer) device. Can only be written.
KBD: The console input device. No character interpretation is performed, and there is no character echo. Again, the input to this device can be redirected, so it does not *always* refer to the physical keyboard.

The MS-DOS device files may also be used (CON, AUX, and PRN without colons ':'). For compatibility reasons, we do not recommend the use of these names.

The MS-DOS 2.xx version of the I/O system will do a search of the default search path (set by the DOS PATH command) if the following conditions are met:

    1) No disk name or path is present in the file name; and
    2) The name is not that of a device.

Alternatively, you may think of the search being done if the file name does not contain any of the characters ':', '/', or '\'.

The default search path cannot be changed while the program is running, as the path is copied by the IntegrAda program when it starts running.

Note:
Creates will never cause a path search as they must work in the current directory.

Upon normal completion of a program, any open external files are closed. Nevertheless, to provide portability, we recommend explicitly closing any files that are used.

Sharing external files between multiple file objects causes the corresponding external file to be opened multiple times by the operating system. The effects of this are defined by your operating system. This is only allowed if all internal files associated with a single external file are opened only for reading (mode In_File), and no internal file is Created. Use_Error is raised if this is violated. A Reset to a writing mode of a file already opened for reading also raise Use_Error if the external file also is shared by another internal file.

Binary I/O of values of access types will give meaningless results and should not be done. Binary I/O of types which are not simple types (see definition in Section F.7, above) will raise Use_Error when the file is opened. Such types require specification of the block size in the form, a capability which is not yet supported.

The form parameter for Sequential_IO and Direct_IO is always expected to be the null string.

The type Count in the generic package Direct_IO is defined to have the range 0 .. 32767.

Ada specifies the existence of special markers called *terminators* in a text file. IntegrAda defines the line terminator to be <LF> (line feed), with or without an additional <CR> (carriage return). The page terminator is the <FF> (form feed) character; if it is not preceded by a <LF>, a line terminator is also assumed.

The file terminator is the end-of-file returned by the host operating system. If no line and/or page terminator directly precedes the file terminator, they are assumed. If the form "Z" is used, the <Ctrl>-Z character also represents the end-of-file. This form is not necessary to correctly read files produced with IntegrAda and most other programs, but may be occasionally necessary. The only

legal forms for text files are "" (the null string) and "Z". All other forms raise USE_ERROR.

If the form is "", the <Ctrl>-Z character is ignored on input. The <CR> character is always ignored on input. (They will *not* be returned by Get, for instance). All other control characters are sent directly to the user. Output of control characters does not affect the layout that Text_IO generates. In particular, output of a <LF> before a New_Page does not suppress the New_Line caused by the New_Page.

On output, the "Z" form causes the end-of-file to be marked by a <Ctrl>-Z; otherwise, no explicit end-of-file character is used. The character pair <CR> <LF> is written to represent the line terminator. Because <CR> are ignored on input, this is compatible with input.

The type Text_IO.Count has the range 0 .. 32767; the type Text_IO.Field also has the range 0 .. 32767.

IO_Exceptions.USE_ERROR is raised if something cannot be done because of the external file system; such situations arise when one attempts:
- to create or open a external file for writing when the external file is already open (via a different internal file).
- to create or open a external file when the external file is already open for writing (via a different internal file).
- to reset a file to a writing mode when the external file is already open (via a different internal file). writing.
- to write to a full disk (Write, Close);
- to create a file in a full directory (Create);
- to have more files open than the OS allows (Open, Create);
- to open a device with an illegal mode;
- to create, reset, or delete a device;
- to create a file where a protected file (i.e., a directory or read-only file) already exists;
- to delete a protected file;
- to use an illegal form (Open, Create); or
- to open a file for a non-simple type without specifying the block size;
- to open a device for direct I/O.

IO_Exceptions.DEVICE_ERROR is raised if a hardware error other than those covered by USE_ERROR occurs. These situations should never occur, but may on rare occasions. For example, DEVICE_ERROR is raised when:
- a file is not found in a close or a delete:
- a seek error occurs on a direct Read or Write; or
- a seek error occurs on a sequential End_Of_File.

Appendix F: Implementation Dependencies

The subtypes Standard.Positive and Standard.Natural, used by some I/O routines, have the maximum value 32767.

No package Low_Level_IO is provided.

F.9   Running the compiler and linker

The IntegrAda compiler is invoked using the following format:

        COMPILE [d:] filename [.ext] {/option}

where filename is an MS/DOS file name with optional disk name [d:], optional extension [.ext], and compiler options {/option}. If no disk name is specified, the current disk is assumed. If no extension is specified, .PKG is assumed.

The compiler options are:

B   Brief error messages. The line in error is not printed (equivalent to turning off pragma VERBOSE).

D   Don't generate debugging code (equivalent to turning off pragma DEBUG)

F   Use in-line 8087 instructions for Floating point operations. By default the compiler generates library calls for floating point operations. The 8087 may be used to execute the library calls. A floating point support library is still required, even though this option is used.

L   Create a listing file with name filename.PRN on the same disk as filename. The listing file will be a listing of only the last compilation unit in a file.

Ld   Create a listing file on specified disk 'd'. Choices are 'A' through 'W'.

Ox   Object code memory model. X is 0 or 1. Memory model 0 creates faster, smaller code, but limits all code in all units of a program to one MS-DOS segment (i.e., 64 kilobytes); Memory model 1 allows code size limited only by your machine and operating system. See the linker (BIND) manual for more information. Memory model 0 is assumed if this option is not given. The compiler records the memory model for which each library unit was compiled, and it will complain if any mismatches occur. Thus, the compiler enforces that if it is run using the /o1 option, then all of the withed units must have been compiled with the same option.

Q   Quiet error messages. This option causes the compiler to not wait for the user to interact after an error. In the usual mode, the compiler will prompt the user after each error to ask if the compilation should be aborted. This option is useful if the user wants to take a coffee break while the compiler is working, since all user prompts are suppressed. The errors (if any) will not stay on the screen when this option is used, therefore, the console traffic should be sent to the printer or to a file. Be warned that certain syntax errors can cause the compiler to print many error messages for each and every line in the program. A lot of paper could be used this way! Note that the /Q option disallows disk swapping, even if the /S option is given.

Rd   Route the JRL file to the specified disk 'd'. Choices are 'A' through 'W'. The default is the same disk as filename.

Sd   Route Scratch files to specified disk. This is useful if you have a RAM disk or if your disk does not have much free space. The use of this option also allows disk swapping to load package specification (.SYM) files. Normally, after both the compiler and source file disks are searched for .SYM files, an error is produced if they are not all found. However, when the /S option is used, the compiler disk may be removed and replaced by a disk to search. The linker has a similar option, which allows the development of large programs on systems with a small disk capacity. Note that disk swapping is *not* enabled by the /S option if the /Q (quiet option) is also given. The /Q option is intended for batch mode compiles, and its purpose conflicts with the disk swapping. The main problem is that when the /S option is used to put scratch files on a RAMdisk, a batch file may stop waiting for a missing .SYM or ERROR.MSG file; such behavior would not be appropriate when /Q is specified.

T   Generate information which allows trimming unused subprograms from the code. This option tells the compiler to generate information which can be used by the remove subprograms from the final code. This option increases the size of the .JRL files produced. We recommend that it be used on reuseable libraries of code (like trig. libraries or stack packages) – that is those compilations for which it is likely that some subprograms are not called.

W   Don't print any warning messages. For more control of warning messages, use the following option.

Wx   Print only warnings of level less than the specified digit 'x'. The given value of x may be from 1 to 9. The more warnings you are willing to see, the higher the number you should give.

X   Handle eXtra symbol table information. This is for the use of debuggers and other future tools. This option requires large quantities of memory and disk space, and thus should be avoided if possible.

Z   Turn on optimization. This has the same effect as if the pragma OPTIMIZE were set to SPACE throughout your compilation.

Appendix F: Implementation Dependencies

The default values for the command line options are:

B   Error messages are verbose.
D   Debug code is generated.
F   Library calls are generated for floating point operations.
L   No listing file is generated.
O   Memory model 0 is used.
Q   The compiler prompts for abort after every error.
R   The JRL file is put on the same disk as the input file.
S   Scratch files are put on the same disk as the compiler.
S   No trimming code is produced.
W   All warnings are printed.
X   Extra symbol table information is not generated.
Z   Optimization is done only where so specified by pragmas.

Leading spaces are disregarded between the filename and the call to COMPILE. Spaces are otherwise not recommended on the command line. The presence of blanks to separate the options or between the filename and the extension will be ignored.

Examples:
        COMPILE test/Q/L
        COMPILE test.run/W4
        COMPILE test
        COMPILE test   .run /B /W/L

The compiler produces a SYM (SYMbol table information) file when a specification is compiled, and a SRL or JRL (Specification ReLocatable or Janus ReLocatable) file when a body is compiled. To make an executable program, the appropriate SRL and JRL files must be *linked* (combined) with the run-time libraries. This is accomplished by running the IntegrAda linker, BIND.

The IntegrAda linker is invoked using the following format:

        BIND [d:] filename {/option}

Here "filename" is the name of the SRL or JRL file created when the main program was compiled (without the .SRL or .JRL extension) with optional disk name [d:], and compiler options {/option}. The filename usually corresponds to the first eight letters of the name of your main program. A disk may be specified where the files are to be found. See the linker manual for more detailed directions. We summarize here, however, a few of the most commonly used linking options:

E   Create an EXE file. This is assumed if the /O1 option is given. This allows
         allow a slightly larger total program size if memory model is used.
F0   Use software floating point (the default).
F2   Use hardware (8087) floating point.
L   Display lots of information about the loading process.
O0   Use memory model 0 (the default); see the description of the /O option in
         the compiler, above.
O1   Use memory model 1.
Q   Use quiet error messages; i.e., don't wait to interact after an error.
T   Trim unused subprograms from the code. This option tells the linker to
         remove subprograms which are never called from the final output file. This
         option reduces space usage of the final file by as much as 30K.

Examples:
         BIND test
         BIND test /Q/L
         BIND test/O1/L/F2

Note that if you do not have a hardware floating point chip, and if you are using
memory model 0, then you generally will not need to use any linker options.

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such
as the maximum length of an input line and invalid file names. A test that
makes use of such values is identified by the extension .TST in its file
name. Actual values to be substituted are represented by names that begin
with a dollar sign. A value must be substituted for each of these names
before the test is run. The values used for this validation are given
below.

Name and Meaning	Value
$BIG_ID1 Identifier the size of the maximum input line length with varying last character.	(1..199 => 'A', 200 => '1')
$BIG_ID2 Identifier the size of the maximum input line length with varying last character.	(1..199 => 'A', 200 => '2')
$BIG_ID3 Identifier the size of the maximum input line length with varying middle character.	(1..100 => 'A', 101 => '3', 102..200 => 'A')
$BIG_ID4 Identifier the size of the maximum input line length with varying middle character.	(1..100 => 'A', 101 => '4', 102..200 => 'A')
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	(1..197 => '0', 198..200 => "298")

Name and Meaning	Value
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	(1..194 => '0', 195..200 => "69.0E1")
$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	(1..100 => 'A')
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	(101..199 => 'A', 200 => '1')
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	(1..180 => ' ')
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	32_767
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	32_767
$FILE_NAME_WITH_BAD_CHARS An external file name that either contains invalid characters or is too long.	<BAD|^^>
$FILE_NAME_WITH_WILD_CARD_CHAR An external file name that either contains a wild card character or is too long.	BAD*.*
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	300_000.0

Name and Meaning	Value
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	1.0E6
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	FROB|IT
$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	<FROB_IT>
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-32768
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	32767
$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	32768
$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-300_000.0
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-1.0E6
$MAX_DIGITS Maximum digits supported for floating-point types.	15
$MAX_IN_LEN Maximum input line length permitted by the implementation.	200
$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	32767
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	32768

Name and Meaning	Value
$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	(1..2 => "2:", 3..197 => '0', 198..200 => "11:")
$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	(1..3 => "16:", 4..197 => '0', 198..200 => "F.E:")
$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	(1 => '"', 2..199 => 'A', 200 => '"')
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.	-32768
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	NOT_APPLICABLE
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFF#

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 27 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form "AI-ddddd" is to an Ada Commentary.

- B28003A: A basic declaration (line 36) incorrectly follows a later declaration.

- E28005C: This test requires that "PRAGMA LIST (ON);" not appear in a listing that has been suspended by a previous "PRAGMA LIST (OFF);"; the Ada Standard is not clear on this point, and the matter will be reviewed by the AJPO.

- C34004A: The expression in line 168 yields a value outside the range of the target type T, but there is no handler for CONSTRAINT_ERROR.

- C35502P: The equality operators in lines 62 and 69 should be inequality operators.

- A35902C: The assignment in line 17 of the nominal upper bound of a fixed-point type to an object raises CONSTRAINT_ERROR, for that value lies outside of the actual range of the type.

- C35904A: The elaboration of the fixed-point subtype on line 28 wrongly raises CONSTRAINT_ERROR, because its upper bound exceeds that of the type.

- C35904B: The subtype declaration that is expected to raise CONSTRAINT_ERROR when its compatibility is checked against that of various types passed as actual generic parameters, may, in fact, raise NUMERIC_ERROR or CONSTRAINT_ERROR for reasons not anticipated by the test.

- C35A03E and C35A03R: These tests assume that attribute 'MANTISSA returns 0 when applied to a fixed-point type with a null range, but the Ada Standard does not support this assumption.

D-1

- C37213H: The subtype declaration of SCONS in line 100 is incorrectly expected to raise an exception when elaborated.

- C37213J: The aggregate in line 451 incorrectly raises CONSTRAINT_ERROR.

- C37215C, C37215E, C37215G, and C37215H: Various discriminant constraints are incorrectly expected to be incompatible with type CONS.

- C38102C: The fixed-point conversion on line 23 wrongly raises CONSTRAINT_ERROR.

- C41402A: The attribute 'STORAGE_SIZE is incorrectly applied to an object of an access type.

- C45332A: The test expects that either an expression in line 52 will raise an exception or else MACHINE_OVERFLOWS is FALSE. However, an implementation may evaluate the expression correctly using a type with a wider range than the base type of the operands, and MACHINE_OVERFLOWS may still be TRUE.

- C45614C: The function call of IDENT_INT in line 15 uses an argument of the wrong type.

- A74106C, C85018B, C87B04B, and CC1311B: A bound specified in a fixed-point subtype declaration lies outside of that calculated for the base type, raising CONSTRAINT_ERROR. Errors of this sort occur at lines 37 & 59, 142 & 143, 16 & 48, and 252 & 253 of the four tests, respectively.

- BC3105A: Lines 159 through 168 expect error messages, but these lines are correct Ada.

- AD1A01A: The declaration of subtype SINT3 raises CONSTRAINT_ERROR for implementations which select INT'SIZE to be 16 or greater.

- CE2401H: The record aggregates in lines 105 and 117 contain the wrong values.

- CE3208A: This test expects that an attempt to open the default output file (after it was closed) with mode IN_FILE raises NAME_ERROR or USE_ERROR; by Commentary AI-00048, MODE_ERROR should be raised.